Feature Interaction Detection in the Feature Language Extensions

Lei Sun*, Lu Zhoa*, Yimeng Li*, Wu-Hon F. Leung**[1]

*Microsoft Corporation, Redmond, WA 98052
**Computer Science, Illinois Institute of Technology, IL 60616

Abstract. One of the most difficult tasks in software development is that features are implemented by changing the code of other features. This problem cannot be solved with existing general purpose programming languages if the features interact and are executed in the same process [1]. A solution to the problem must include a method that can automatically identify where to make the changes, or in the context of interacting features, automatically detect their interaction conditions. The Feature Language Extensions (FLX) is a set of language constructs that enable the programmer to develop interacting features as separate and reusable program modules. This paper overviews the feature interaction detection method of FLX. It includes an algorithm that determines the satisfiability of quantifier free first order predicate formulas containing variables that are used in the actual software and, therefore, may have complex data structures and predicate methods. FLX provides language constructs for the programmer to specify reusable predicate combination functions so that the algorithm does not require iterations of trials and errors.

Keywords: Feature interaction, program entanglement, feature interaction detection, satisfiability of first order formulas

1 Introduction

One of the most difficult tasks in software development is that features are implemented by changing the code of other features. It is laborious and error prone. Programmers must examine code very carefully to determine where to make the changes; regression testing, which is costly and time consuming, must be carried out.

The term *feature* is used to denote certain functionality of an application. For example, reliable data transport and congestion control are two features of the Internet protocol TCP. In software engineering literature, terms like *features*, *aspects* and *concerns* are used interchangeably. When a feature is implemented by changing the code of another feature, the programs for these two features *entangle* in the same reusable program unit of the programming language. Entangled features are also difficult to understand, maintain and reuse.

[1] Corresponding author: Wu-Hon Francis Leung, Computer Science, Illinois Institute of Technology, 10 West 31st Street, Chicago, IL 60616. email: leung@iit.edu

If two features *interact* (C1), they are executed in the same process (C2), and they are implemented with a programming language that requires the programmer to specify execution flow (C3), then their programs will inevitably entangle [1]. If they do not interact, their programs do not have to entangle. Two features interact if their *behaviors* change when integrated together. Since a feature is implemented by a computer program, its behavior is manifested in the sequence of statements that gets executed and the value of its output for a given input. We do not consider two features to be interacting if one merely changes the input to the other but not its behavior.

Interacting features are common place. For example, normal processing and exception handling features interact: before integration with exception handling features, normal processing features will crash when an exception occurs (C1). These two kinds of features often need to be executed in the same process (e.g. when the exception handling feature must immediately stop normal processing when an exception occurs) (C2). Existing general purpose programming languages require (C3). With these languages, the two kinds of features always entangle.

The *interaction conditions* between two interacting features are the conditions under which behavior change will occur. An interaction condition is *resolved* with specification of the changed behavior. Presently, the programmers must trace the different execution sequences and reason on the value of variables to identify where in the code the interaction conditions will become true, and change code to resolve the interaction. A solution to the entanglement problem therefore needs to meet these requirements: A feature can be developed as a reusable program module independent of its interacting features (R1). The interaction conditions among interacting features can be detected automatically (R2). The interaction can be resolved without requiring code changes to the features (R3).

The Feature Language Extensions (FLX) is a set of programming language constructs designed to meet these requirements. It supports *nonprocedural* programming. A program unit consists of a *condition part* and a *program body part.* The program body gets executed when its corresponding condition part becomes true; the programmer does not specify execution flows thus relaxing (C3). The manner in which FLX meets (R1) and (R3) are described in [1] and [2] and briefly reviewed in Section 2 and 3. Meeting (R1) is sometimes called feature oriented programming (FOP). Previous attempts to FOP (e.g. [11]. [12] and [5]) do not meet (R3). The focus of this paper, however, is our method to meet (R2).

The problem of interaction detection has been studied with a variety of formalisms such as temporal logic [3] and Petri Nets [4]. Particularly notable are those that demonstrated interaction detection mechanically on applications implemented with the specification languages of existing model checkers (e.g. [5] and [6]). A review of several interaction detection methods is given in [7]. Formalisms and specification languages do not handle variables with complex data structures and predicate methods found in actual software. The interaction detection method of FLX handles them.

At the heart of our feature interaction detection method is an algorithm (first order satisfiability solver) that determines the satisfiability of a quantifier free first order formula whose variables come from FLX programs[2]. Existing algorithms typically

[2] A logical formula is satisfiable if there is a value assignment to its variables that will make it true.

use search and bound strategies that involves iterations of solving NP complete problems. A review of the state of art can be found in [9]. FLX provides language constructs for the programmer to specify the *predicate combination functions* of the essential variables in the formula and avoids the search and bound.

A research version of FLX to Java compiler exists; it adds FLX constructs to Java similar to C++ added object oriented constructs to C. About forty features and feature packages had been written in FLX for a telephony system to test the compiler. The compiler and the telephony system can be downloaded from [10]. More recently, we used FLX to develop a call center over Skype. The programming language support for the first order satisfiability algorithm that will be described later is a new version and has not been implemented yet at the time of writing this paper.

In the rest of the paper, FLX is introduced in Section 2. The interaction detection method of FLX is described in Section 3. Section 4 concludes.

2 A Brief Introduction to FLX

FLX is designed for the development of feature rich components called *feature packages*. In a telephony system developed with FLX, a telephone object is associated with two feature packages: one for call processing features such as call forwarding and the other for digit analysis features such as speed calling. A feature package integrates and resolves the interaction among a set of *features*. Features and feature packages are reusable. Different combinations of them can be integrated into different feature packages. Therefore different telephones can have different call processing and digit analysis features.

A *feature* is composed of a set of nonprocedural program units. It is designed according to a *model* instead of the code of other features. The model is composed of a *domain statement* and an *anchor feature*. The domain statement specifies the condition variables that will be used in the condition part of a program unit. The anchor feature provides the basic functionality. Other features that refer to it can be considered as its enhancements or extensions.

```
 anchor feature Pots {
   domain BasicTelephony;
MakeCall {
   condition: state.equals(State.IDLE);
   event: Offhook; {
      fone.applyDialTone();
      state = State.DIALING;
      }
   }
ReceiveCall {
   condition: state.equals(State.IDLE);
   event: TerminationRequest e; {
      Ringing r = new Ringing(e.FromPID);
      rt.sendEvent (r);
      state = State.RINGING;
      }
   }
}
```

Figure 1. A Portion of the POTS code

```
feature DoNotDisturb {
domain  BasicTelephony;
anchor  POTS;
Router rt;

SayBusy {
   condition: all;
   event: TerminationRequest e; {
      Busy b = new Busy(e.FromPID);
      rt.sendEvent (b);
      }
   }
}
```

Figure 2. The feature `DoNotDisturb`

Figure 1 shows two of the program units in the anchor feature **POTS** (plain old telephone service). The program unit **MakeCall** specifies that when the phone is idle and the software receives an **Offhood** signal, dial tone is applied to the phone and the state of the phone is changed to **DIALING**. The domain statement, **BasicTelephony**, is not shown and can be found in [1].

Figure 2 shows the feature **DoNotDisturb** which returns busy to all callers (who send the **TerminationRequest** message). The **CallForwarding** feature with its most important program unit is given in Figure 3. The feature forwards the call if a call forwarding number has been specified and the number is not the same as the caller.

```
feature CallForwarding {
domain: BasicTelephony;
anchor: POTS;

ForwardCall {
    condition: state.equals (State.IDLE);
    event: TerminationRequest e; {
       if ((forwardNumber != “”) && (forwardNumber != e.fromPID)) {
              rt.send (forwardNumber, e);
              stop;
              }
       }
     }
}
```

Figure 3. The ForwardCall program unit of CallForwarding

While **DoNotDisturb** and **CallForwarding** both depend on **POTS**, they can be developed independent of one another. They can be reused in the same or different feature packages (see examples in [1]). It is in this sense that they meet (R1).

Figure 4 shows one way that **DoNotDisturb** and **POTS** can be integrated in a feature package called **QuietPhone**. The two features interact whenever the phone is called. The interactions are resolved in a priority precedence list which specifies that when an interaction condition becomes true, only the program unit belonging to the feature with the highest precedence will get executed. In this case, whenever the phone is called, only the program unit **SayBusy** of **DoNotDisturb** is executed. However, the **DoNotDisturb** feature does not block the user from making calls.

```
feature package QuietPhone {
  domain: BasicTelephony;
  features: DoNotDisturb, POTS;

  priorityPrecedence (DoNotDisturb, POTS);
}
```

Figure 4. The QuietPhone feature package

DoNotDisturb and **POTS** are not modified in **QuietPhone** therefore meeting (R3). In [1] and [2] we show other ways the three features may be integrated. Precedence lists are not the only facility provided by FLX to resolve feature interaction. In fact, we show in [2] that they are insufficient in some situations. But they are very useful; they can resolve many interaction conditions in a single statement.

A feature package is compiled into a Java class. An object, **x**, instantiated from the feature package class is invoked by a generated interface method **x.sendEvent(e)** where **e** is an event defined in the domain statement of the feature package.

3 Feature Interaction Detection in FLX

We will describe the feature interaction detection method of FLX in three parts. First, we explain how we know the interaction condition among programs written in FLX. In 3.2, we describe the input that the FLX algorithm needs from the programmers and the FLX language facilities that enable the programmers to do so. Finally, we give the basic FLX first order satisfiability algorithm in 3.3.

3.1 Interaction Conditions

The condition part of a program unit written in FLX is composed of a *condition statement* and an *event statement*. The condition statement is a Java condition expression: it is a Boolean formula of predicate methods and Boolean variables. More formally, it is called a quantifier free first order formula. In object oriented programming, each predicate method operates on a variable (or object). FLX requires that such variables must be declared in a domain statement and are called *domain variables*. In a quantifier free formula, universal quantification is implied. The existential quantifier is not supported. When the programmer has the need to say something like “there exists some elements”, we ask him to write a predicate method **non-empty()** instead. For the purpose of this paper, the event statement specifies a list of events declared in the domain statement.

A condition part becomes true if the feature package receives an event specified in its event statement and its condition statement is true at that time. Its corresponding program part, which is a Java block, gets executed and the triggering event is then consumed. A feature package is executed in a process. The execution of a program unit is *atomic*, meaning that the process must carry the execution of the program unit to completion before executing another program unit of the feature package.

Under these conditions, if the condition parts of two program units can become true at the same time, or equivalently if the conjunction of their condition part is satisfiable, there is ambiguity on which one of them should be executed. We say the program units *interact* with one another, as the behavior of one of them or both must be changed to resolve the ambiguity. If these two program units belong to the same feature, FLX requires that their interaction, or the ambiguity, must be resolved before the feature is compiled. The interaction can be resolved by rewriting the program units such as changing their condition parts.

If the two interacting program units belong to two different features, the two features interact. The satisfiable condition of the conjunction of their condition parts is an interaction condition of the two features. FLX requires that when two features interact, their interaction conditions must be resolved in a feature package.

3.2 The Domain Data Types of FLX

FLX requires that a domain variable must be of a *domain data type*. A domain data type is derived from a Java class by associating the Java class with a *combination class*. A combination class defines a set of predicate methods and a combination function that when given a list of the predicate method will return the decision of whether their conjunction is satisfiable.

Consider an application (such as the code generator of a compiler) whose features are invoked depending whether a node precedes or follows other nodes in a directed acyclic graph. The class **GNode** shown in Figure 5 is an example of such a node. Since the relationship among nodes in a directed acyclic group follows a partial order[3], we choose to associate **GNode** with the **PartialOrder** combination class to derive the domain data type, **DGNode** (Figure 6), for the application.

```
public class GNode {
    private List <GNode> parents;
    private List <GNode> children ;
    …
    public GNode (List <GNode>, List <GNode>) {…}
    public insertChild (GNode) {…}
    …
}
```

Figure 5 GNode declaration (partial)

```
public domain class DGNode
    associates GNode
    with PartialOrder;
```

Figure 6 Declaration of the domain data type DGNode

A combination class is often declared as a generic class so that it can be associated with different Java classes. In this case, the **PartialOrder** combination class (Figure 7), can be associated with integers, sets and other data structures. The decision procedure for the combination function of **PartialOrder** is quite straight-forward: if there exist predicates that contradict one another in a partial order (e.g. we have a.precedes (b) and b.precedes (a)), the combination function will return false otherwise it returns true. Efficient algorithms to do so are well known. We have implemented a number of different combination classes. They range from those that are associated with Java primitive types such as **int** to those that are associated with Java classes in the collection framework.

```
public combination PartialOrder <E extends Comparable <E>>
{
    E element;
    public boolean precedes (E e) { … }
    public boolean follows (E e) { … }
    public boolean equals (E e) { … }
    public combinationFunction (HashSet <String> group) {…}
}
```

Figure 7 Declaration of the combination class PartialOrder

The combination class **PartialOrder** contains the three expected predicate methods. FLX requires that predicate methods in a combination cannot have the side effect of modifying the domain variables. In other words, an FLX first order formula is also function free. The FLX compiler checks that domain variables do not appear in the left hand side of an assignment statement in a combination class.

The domain data type **DGNode** inherits from both **GNode** and **PartialOrder** in the sense that the methods implemented in the latter two classes are available to objects of **DGNode**. But the programmer can use only the predicate methods defined in the combination class in the condition statement of a program unit.

[3] A partial order is a reflexive, antisymmetry and transitive relation between two elements of a set P, denoted by "≤". For all a, b and c in P, we have (i) $(a \leq a)$ (reflexivity); (ii) if $(a \leq b)$ and $(b \leq a)$ then $(a = b)$ (antisymmetry); and (iii) if $(a \leq b)$ and $(b \leq c)$ then $(a \leq c)$ (transitivity).

3.3 The Basic FLX First Order Satisfiability Solver

The basic algorithm involves three steps: First, the first order formula is converted into its Disjunctive Normal Form (DNF). Each clause of the DNF is a conjunction of literals[4]. Taking advantage of the associative property of the conjunction operator, we partition each clause into subgroups in step 2. Each subgroup contains literals whose domain variables are of the same domain data type. In step 3, we pass each subgroup to its corresponding combination function. If the combination function returns false, the subgroup and therefore the clause is not satisfiable. If all the clauses in the DNF are not satisfiable, the formula is not satisfiable. If any of the clauses is satisfiable, the formula is satisfiable. The algorithm does not stop after finding one clause is satisfiable because it also needs to identify the total satisfiable space. Therefore, it will check all the clauses in the DNF and identify all those that are satisfiable. The union of satisfiable clauses constitutes the satisfiable space.

The above algorithm differs from contemporary satisfiability solvers, including those designed only for Boolean formulas, on two counts: First, other solvers transform the logical formula to its conjunctive normal form (CNF) instead of DNF. Second, other solvers used search strategies to look for satisfiable assignments; we ask the programmer to supply a decision procedure.

Our motivation on the second count came from the observation that existing solvers know very little about the predicates, but in software, the programmers know exactly how to interpret the predicates and how they relate to one another. What is needed is to find some way for the programmers to input their knowledge to the solver, and we choose the combination function class as the vehicle. On the first count, we could have used a CNF and ask the programmer to specify a procedure that will return whether the disjunction of a list of predicates is satisfiable. We found it easier to specify the combination function. We considered a fair number of representative data types from the Java library, in all cases their combination function classes can be implemented straightforwardly and efficiently.

Converting a logical formula to its DNF is NP-complete. The satisfiability solvers for Boolean formulas (SAT solvers) are also NP-complete but they represent an enabling technology in the real world for Electronic Design Automation (EDA) and some showed the ability to solve problem instances involving tens of thousands of variables [13]. In our case, the number of variables is confined to those defined in a domain statement.

4 Conclusions

Any solution to the program entanglement problem should solve the interaction detection problem. The approach taken by FLX first takes advantage of the fact that FLX programs are nonprocedural programs and the interaction condition between two interacting program units is the satisfiable condition of their condition parts. Secondly, FLX provides programming language facilities so that the programmer can provide

[4] A literal is either an atom or its negation in a logical formula. In a first order formula, an atom is either a Boolean variable or a predicate.

the decision procedure embodied in the combination functions to reduce the complexity of determining the satisfiability of first order formulas whose variables come directly from the software. Combination function classes are reusable. Maybe in the future there will be a library of them to meet most of the programmers' needs.

We are working on a new version of the FLX first order satisfiability solver and its associated language facilities. The algorithm is also at the heart of the FLX compiler code generator. An efficient SAT solver is an essential component of electronic design automation (EDA) that includes model checking and formal verification [13]. We do not have corresponding tools for software design because in hardware design assertions are Boolean formulas as their variables are binary variables, but in software one must reason on predicate logic and ask questions like whether a linked list is empty. We are now looking into utilizing the FLX first order satisfiability solver to enable automatic verification of FLX programs according to assertions, instead of relying on case by case testing.

References

1. Leung, W. H.: Program Entanglement, Feature Interaction and the Feature Language Extensions. Computer Networks, Volume 51, February, 2007, 480-495
2. Yang, L., A. Chavan, K. Ramachandran, W. H. Leung: Resolving Feature Interaction with Precedence Lists in the Feature Language Extensions, Proceedings of International Conference on Feature Interaction, IOS Press, 2007.
3. Felty, A. P., K. S. Namjoshi: Feature Specification and Automated Conflict Detection, Proceedings of Feature Interactions in Telecommunication Systems, IOS Press, 2000.
4. Nakamura, M., Y. Kakuda, T. Kikuno: Petri-net Based Detection Method fro Non-deterministic Feature Interaction and Its Experimental Evaluation, Proceedings of 3rd International Workshop on Feature Interactions in Telecommunication Systems, IOS Press, 1997.
5. Plath, M., M. D. Ryan: The Feature Construct for SMV: Semantics, Proceedings of Workshop on Feature Interactions in Telecommunication Systems, IOS Press, 2001.
6. Calder, M., A. Miller, Using SPIN for Feature Interaction Analysis – A Case Study, Proceedings of SPIN2001, 2001.
7. Calder, M., M. Kolberg, E. Magill, S. Reiff-Marganiec: Feature Interaction: A Critical Review and Considered Forecast, Computer Networks, Vol. 41, January, 2003.
8. Areces, C., W. Bouma, M. de Rijke: Feature Interaction as a Satifiability Problem, Proceedings of Feature Interaction in Telecommunication and Software Systems, IOS Press, 2000.
9. Zhao, L.: A First Order Satisfiability Solver for the Feature Language Extensions, M.S. these, ECE Department, IIT, May, 2006.
10. www.openflx.org
11. C. Prehofer, An object oriented approach to feature interaction, in: Proceedings of the Feature Interaction Workshop,1997, IOS Press.
12. D. Batory, J.N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, in: Proceedings of International Conference on Software Engineering 2003 (ICSE 2003), Portland, Oregon,May 2003.
13. http://en.wikipedia.org/wiki/Boolean_satisfiability_problem